**THE ABSTRACTION**

When considering the appropriate approach to implement this project, the primary goal is to ensure it is easy to maintain, extend, and test. To achieve this, the project will be structured into three layers: the user interface layer, the domain layer, and the data access layer.

The user interface layer, as its name suggests, is responsible for providing the interface through which users interact with the application. However, an equally important task of this layer is to define the abstractions that the domain layer will implement. These abstractions serve as the requirements for components, enabling them to work with data. Therefore, the domain layer is where the definitions of these abstractions, originating from the user interface layer, are established. When it comes to extending the original abstraction, there are only a few potential changes that may occur. These changes include:

1. The return type of the object when the concrete implementation of the abstraction is modified. This change implies that the object model used by the components in the user interface layer has also changed. It could involve altering the requirements, requiring more or less information, or even necessitating a completely new object model.

2. The data provided for request data undergoes changes, which is expected to evolve over time. Because of that, the abstraction needs to be something that is easy to modify, while also providing sufficient information to define how the abstraction should function. That is the reason why I have designed it this way:



The concept underlying this design is to create a generic abstraction, which serves as the communication bridge between the UI layer and Domain layer, capable of accepting various types of input and output for data requests. It is essential for the domain layer to understand the nature of the requested data and the required input for data retrieval. However, a challenge arises in determining the appropriate location for defining the input model.

In this design, I have decided to define the input model within the requester itself. This approach ensures that the UI layer does not need to rely on input models defined in the domain layer, over which it has no control, to request data. Nonetheless, this solution does not address the issue of the requester's inability to provide specific requirements regarding the input data. This poses a significant problem when the input model in the UI layer cannot fulfill the needs of the requester in the domain layer.

To resolve this, I believe that the domain layer, being responsible for defining how the abstraction should function, should have knowledge of the appropriate input model to interact with. The domain layer has the authority to decide whether the requester implements the abstraction from the UI layer or not.

I will explain why I want request return IEnumerable<TOutput> below.

Moving forward with the abstraction from the user interface layer, the next consideration is how the requester of the domain class should implement this abstraction. So I decided to design according to this approach:  


However, the most significant challenge lies in designing the data model that satisfies the needs of all stakeholders involved. The data model should address two key aspects:  
Requester Dependency for Data Retrieval: The requester requires a dependency that can retrieve data from the server. This dependency needs to understand how to request data from the server, convert the returned data into a format compatible with the requester and repository data models, and handle notifications in case the requested data is not returned.  
Requester Dependency for Data Storage and Updates: The requester also requires a dependency that can handle data storage, updates, and removal. This dependency should be capable of saving and updating data, as well as handling data retrieval.

However, does the requester really need to have two dependencies? Is it feasible, instead of just returning the requested data, to have an additional dependency (decorator dependency) that performs other tasks? Yes, that is entirely possible. Therefore, I want to design the abstraction in such a way that regardless of the approach, the definition of a specific requester is not bound to a single dependency approach. Because, regardless of the approach, the sole task of a specific requester is to accurately identify the dependency it needs to use when injecting dependencies, nothing more, nothing less.



The responsibility of the domain layer is to provide the capability to query and update data for the definition of requester. Therefore, two interfaces, IQuery and ICommand, are provided for repositories to use. These interfaces enable the repositories to change their behavior at runtime, allowing a repository to be instantiated a limited number of times and enabling dependency injection with the interfaces.

The final task is to design how data is sent and received between two distinct entities, such as the client and server. The idea for this design comes from thinking about how a restaurant handles customer requests. Imagine, for example, a pho restaurant where a customer requests something different. It could be a request for an extra bottle of fish sauce to add to the pho or a request for the pho to be served on a plate. To fulfill the request, the restaurant can provide the requested bottle of fish sauce and inform the customer that they cannot serve the pho on a plate while preserving the essence of the dish. The customer will then handle the situation based on the restaurant's response. And that's the kind of operation I want. The requester can make any request, but the handler can choose to process it or communicate that it cannot be handled. In the context of designing data communication between the client and server, there are certain aspects to consider: the data format for communication, the desired format for data transmission and reception, and the value of the data being sent and received. Specifically, the sender will transmit the data in a specific textual format, such as JSON or XML, and include a message to indicate its intent and facilitate appropriate processing by the receiver. Additionally, the sender needs to provide instructions regarding the expected data format for reception, including information about the type name and the names of the data-containing attributes (note that this data type should function as a struct, even if it is implemented as a class). Lastly, the sender can make requests regarding the data, similar to how it requests the data format for the response. Finally, the receiver's task is to strive to fulfill all the requests made by the sender and have a default notification in case it cannot fulfill a request.



The main challenge that this process needs to address is how to identify and find suitable handlers for the requests. Specifically, what information is necessary to accurately determine the appropriate handler for a given conditional object, or what information is needed to determine if any handler can return data of a specific type after processing. Ultimately, the information being sent and processed must adhere to a common format, meaning that the information being sent does not explicitly specify the exact data type that the sender/handler must possess.

Another challenge in handling requests is analogous to how a restaurant deals with specific customer requests. How do we determine if a request is indeed specific? If a request falls outside the realm of what the restaurant can accommodate, does it still qualify as a specific request? What if the request is similar to an existing dish, albeit with some minor differences? Do all approaches become indistinguishable when we only focus on the conditions and outcomes?

To address these questions, we must consider a fundamental issue: can a piece of code effectively address this problem? It parallels the predicament faced by a restaurant when confronted with an unusual request. For example, imagine a sandwich establishment that does not serve hotdogs. What should they do if a customer insists on ordering a hotdog? In the current scenario, it's possible that the condition object being sent may share the same name but have different properties compared to the corresponding handler. Alternatively, they might have the same properties but different names. There could even be instances where the type and property names are identical, but their property values do not align.

In such situations, it becomes increasingly challenging for the code to determine which handler should be employed, as there are numerous considerations to account for. It is unlikely that these complexities can be managed automatically. Thus, the optimal approach is to disregard the request, throw an exception, or take similar actions. Just as a hotdog restaurant cannot serve a specific type of hamburger due to unfamiliarity or lack of appropriate ingredients, it is impossible to guarantee that attempting to create a hamburger using a hotdog recipe will yield the same taste and flavor. This analogy applies when contemplating how to handle requests. The process of making a hotdog or hamburger directly impacts its taste. Consequently, the most prudent course of action in such scenarios is to ignore requests that cannot be accommodated.

Because of that, this solution for this situation is the same with above: using DI container to contain object that handle request. This DI container use the pair of condition object and result object to identify handler. Those handlers will follow the Query - Command Principle. With Command type object, there is a specific type that will be called default to identify that is a Command type object.

The last problem is how to make a manager for all of these stubs.